

A distributed resource allocation algorithm for many processes

Wim H. Hesselink (whh469)
University of Groningen, The Netherlands
w.h.hesselink@rug.nl

March 3, 2013

Abstract

Resource allocation is the problem that a process may enter a critical section *CS* of its code only when its resource requirements are not in conflict with those of other processes in their critical sections. For each execution of *CS*, these requirements are given anew. In the resource requirements, levels can be distinguished, such as e.g. read access or write access. We allow infinitely many processes that communicate by reliable asynchronous messages and have finite memory. A simple starvation-free solution is presented. Processes only wait for one another when they have conflicting resource requirements. The correctness of the solution is argued with invariants and temporal logic. It has been verified with the proof assistant PVS.

Key words: distributed algorithms; resource allocation; drinking philosophers; readers/writers problem; verification; starvation freedom; fairness

1 Introduction

Resource allocation is a problem that goes back to Dijkstra's dining philosophers [9] and the drinking philosophers of Chandi and Misra [4]. It is the problem that a process may enter a critical section of its code only when its resource requirements are not in conflict with those of other processes in their critical sections. In the case of the dining philosophers, the philosophers form a ring and the resource requirements are two forks shared with the neighbours in the ring. In the drinking philosophers problem the philosophers form an arbitrary finite undirected graph.

In the general resource allocation problem, there is a number of processes that from time to time need to execute a critical section *CS* in which they need access to some resources. For every critical section of every process, the resource requirements may be different. Processes that are concurrently in a critical section, must have compatible resource requirements. The processes must therefore communicate with possible competitors, and possibly wait for conflicting processes before entering *CS*. On the other hand, unnecessary waiting must be avoided.

1.1 Setting and sketch of solution

We present a solution for a setting with infinitely many processes that have private memory and communicate by asynchronous messages. The processes and messages are assumed to be reliable: the processes never crash, the messages are guaranteed to arrive and be handled, but the delay is unknown. Messages are not lost, damaged, or duplicated. They can pass each other, however, unlike in [3, 19] where the

messages in transit from one sender to one receiver are treated first-in-first-out. Every process can send messages to every other process, and receive messages from it. The processes receive and answer messages even when they are idle.

The resource requirements can be sets of resources the processes need exclusive access to. More generally, however, resource requirements may comprise, e.g., read access or write access to some data, where concurrent reading is allowed while writing requires exclusive access.

In our solution, we deal with the infinitely many processes by splitting the problem in two parts: one part to ensure that a process has, for every job, only a finite set of potential competitors, its dynamic neighbourhood, the other part to use this neighbourhood to ensure partial mutual exclusion.

The two parts interact mildly. We call the first part the *registration algorithm*, because it is based on the idea that processes need to register for resources. The second part is called the *central algorithm* because it satisfies the functional requirements. If the neighbourhoods can be kept constant, the registration algorithm can be removed, and the central algorithm can be compared with the drinking philosophers [4].

When a process gets a new job, it first registers to obtain a list of potential competitors. If in doing this it extends its registrations, it may need to contact some of these competitors before proceeding. Then, the central algorithm takes over. This first lets the process wait for conflicting processes that are currently competing for CS, and then allows it in the competition for CS. A process with many registrations has much communication to perform. We therefore offer the processes the option to withdraw registrations, in some degree concurrently with resource acquisition.

The algorithm has two kinds of waiting conditions: waiting for messages in transit to arrive, and waiting for conflicting processes to proceed. It is not our aim to minimize the waiting time. We offer a simple solution with as much nondeterminism as possible and as much progress as we can accommodate in view of the safety requirements.

1.2 Overview and verification

We briefly discuss related research in Section 1.3. In Section 1.4, we describe the model and the notations for message passing.

Section 2 presents the algorithms. Section 3 contains its proof of safety. In Section 4, we introduce and formalize progress, and prove that weak fairness guarantees progress for the registration algorithm. In Section 5, we define, formalize, and prove progress of the central algorithm in a form that combines starvation freedom and concurrency. We discuss message complexity and waiting times in Section 6, and conclude in Section 7.

The proofs of the safety and liveness properties have been carried out with the interactive proof assistant PVS [20]. The descriptions of proofs closely follow our PVS proof scripts, which can be found on our web site [12]. It is our intention that the paper can be read independently, but the proofs require so many case distinctions that manual verification is problematic.

1.3 Related research

The readers/writers problem [2] goes back to Courtois, Heymans, and Parnas [7], in the context of shared memory systems with semaphores. We are not aware of solutions for systems with message passing.

In the drinking philosophers' problems of [4, 19, 24], the philosophers form a fixed finite undirected graph. In this case, the set $nbh.p$ of possibly conflicting

processes of a process p is a subset of the constant set $Nbh.p$ of p 's neighbours in the graph. This subset is chosen nondeterministically when the process becomes “thirsty”. The message complexity of the solutions in these papers is proportional to the size of $Nbh.p$ and not to the possibly considerably smaller size of $nbh.p$. This is a disadvantage for cases with large complete graphs. To enforce starvation freedom, these solutions assign directions to the edges of the graph such that the resulting directed graph is acyclic.

Much work has been done to minimize the response time [3, 6, 21, 22, 24]. For instance, the paper [24] offers the possibility of a waiting time that is constant and not proportional to the (in our case unbounded) number of processes. It does so by means of the algorithm of [18], which uses a linear ordering of the resources adapted to a fixed network topology. The papers [22, 24] offer modular approaches to the general resource allocation problem.

Another important performance aspect is robustness against failures. The paper [6] introduces the measure of failure locality, see also [23]. The paper [8] concentrates on self-stabilization, while imposing specific conditions on the resource requirements.

As far as we can see the only papers that treat the dynamic resource allocation problem that allows conflicts between arbitrary pairs of processes are [3, 22]. The algorithm of [3] ignores the resources and takes the conflicts as given. Whenever two processes have conflicting jobs, at least one of the two is activated with knowledge of the conflict. The emphasis of [3, 22] is on minimizing the response time. The algorithms are more complicated and need more messages than our solution. The paper [22] uses resource managers.

1.4 Asynchronous messages

The processes communicate by reliable asynchronous messages, i.e., the messages are not corrupted, lost, or duplicated. They may, however, pass each other.

Every message has a message key, a sender, and a unique destination. It may have a value. As in CSP [13], we write $m.q.r!$ for the command for q to send a message with key m to destination r , and $m.q.r?$ for the command for r to receive this message. Unlike CSP, the messages are asynchronous. We write $m.q.r!v$ for the command to send a message with key m and value v from q to r , and $m.q.r?v$ for the command to receive message m and assign its value to v .

In the algorithm, for every message key m , and for every source q and destination r , there is never more than one message in transit from q to r . Therefore, e.g., in Promela, the language of the model checker Spin [14], one could model the messages by channels with buffer size 1.

For the correctness of the algorithm, the time needed for message transfer can be unbounded. For other issues, however, it is convenient to postulate an upper bound Δ for the time needed to execute an atomic command plus the time that the messages sent in this command are in transit. Similarly, when discussing progress, we assume that the execution time of the critical sections is bounded by Γ .

2 The Algorithm

Section 2.1 contains the functional specification of the complete algorithm, and separates the responsibilities of the central algorithm and the registration algorithm. Progress requirements are discussed in Section 2.2.

The central algorithm is sketched in Section 2.3. Section 2.4 contains the code of the central algorithm and discusses some global aspects. In Section 2.5, we present its design as a layered algorithm.

In Section 2.6, we develop a job model as a preparation for the registration algorithm presented in Section 2.7. Section 2.8 contains commands to abort the entry protocol.

The entire algorithm is presented in this section without verification. The verification is postponed to Section 3. Yet, we designed the algorithm concurrently with the verification, because that is for us the only way to obtain a reliable algorithm. We separate the two aspects here for the ease of reading.

2.1 Functional specification

When a process gets a new job to execute in a critical section, the associated resource requirements are chosen nondeterministically, say by a (distributed) environment. We use the term *job* for these resource requirements, and we use the type *Job* for all possible jobs. The value *none* : *Job* represents the absence of resource requirements. We give every process *q* a private variable *job.q* : *Job*, which is initially *none*, but which is modified by the environment when it gives the process a job.

Processes concurrently in the critical section *CS* must have *compatible* jobs. We write $u * v$ to express that the jobs *u* and *v* are compatible. We assume that compatibility satisfies the axioms that $u * \text{none} \equiv \text{true}$ and $u * v \equiv v * u$ for all jobs *u* and *v*. Examples of compatibility relations are given in Section 2.6.

The problem of *resource allocation* is thus to ensure that processes with *job* \neq *none* eventually enter *CS*, under the safety requirement that, when two different processes are both in *CS*, their jobs are compatible:

$$Rq0: \quad q \text{ in } CS \wedge r \text{ in } CS \Rightarrow q = r \vee job.q * job.r .$$

Here and henceforth, *q* and *r* stand for processes. For all invariants, we implicitly universally quantify over the free variables, usually *q* and *r*. If *v* is a private variable, outside the code, the value of *v* for process *q* is denoted by *v.q*.

We speak of a *conflict* between *q* and *r* when *q* and *r* have incompatible jobs: $job.q * job.r \equiv \text{false}$. Clearly, the conflict relation is time dependent. Condition *Rq0* says that conflicting processes are never concurrently in their critical sections. For comparison, mutual exclusion itself would be the requirement that $q \text{ in } CS$ and $r \text{ in } CS$ implies $q = r$.

While we allow potentially infinitely many processes, we cannot expect a process to communicate with infinitely many competitors. We therefore introduce a *registration algorithm* that uses the new job of a process *q* to provide *q* with a finite set *nbh0.q* of potential competitors. Every process not in *nbh0.q* must not in conflict with *q*. In other words, the registration algorithm serves to guarantee the invariant:

$$Rq1: \quad q \text{ in } CS \wedge r \text{ in } CS \Rightarrow q = r \vee r \in nbh0.q \vee job.q * job.r .$$

The *central algorithm* uses the sets *nbh0.p* to guarantee the invariant:

$$Rq2: \quad q \text{ in } CS \wedge r \text{ in } CS \wedge r \in nbh0.q \wedge q \in nbh0.r \Rightarrow job.q * job.r .$$

Predicate *Rq0* follows from *Rq1* and *Rq2*, using the symmetry of the operation $*$. The reason for the name *nbh0* is that the algorithm also uses a closely related variable *nbh* which does not satisfy *Rq2*.

As a process may have to wait a long time before it can enter *CS*, it may be useful that the environment of a process can nondeterministically abort its entry protocol and move it back to the idle state. This option is offered in Section 2.8.

2.2 Progress requirements

The first progress requirement that comes to mind, is *starvation freedom* [10], also called lockout freedom [19]. This means that every process that needs to enter *CS*, will eventually do so unless its entry protocol is aborted.

While starvation freedom is important, resource allocation has a second requirement, viz. that no process is hindered unnecessarily. This property is called *concurrency* in [4, 22]. It means that every process that needs to enter *CS* and does not abort, will eventually enter *CS*, unless it comes in eternal conflict with some other process (a kind of deadlock).

Of course, concurrency follows from starvation freedom. We introduce both, however, because concurrency needs a weaker liveness assumption than starvation freedom. The liveness assumptions needed are forms of weak fairness. Weak fairness for process p means that if, from some time onward, process p is continuously enabled to do a step different from aborting, it will do the step. Weak fairness for messages m from q to r means that every message m in transit from q to r will eventually arrive. Weak fairness is a natural assumption, and some form of it is clearly needed. We come back to this in Section 4.1.

Our algorithm satisfies starvation freedom under the assumption of weak fairness for all processes and all messages. Concurrency for process p only needs weak fairness for p itself and weak fairness for all messages from and to p . The point is that progress of process p is not hindered by unfair processes without conflicts with p ; e.g., such processes are allowed to remain in *CS* forever. In Section 4.4, we unify starvation freedom and concurrency in a single progress property called absence of localized starvation.

2.3 Sketch of the central algorithm

The central algorithm thus works under the assumption that, for every *CS* execution, the process obtains a finite set *nbh* of potential competitors with the guarantee that the *CS* execution does not lead to a conflict with other processes. The elements of *nbh* are called the neighbours of the process. We do not yet distinguish *nbh* and *nbh0*.

Inspired by the shared-variable mutual exclusion algorithm of [17], the central algorithm is designed in three layers: an outer protocol to communicate the job to all neighbours, a middle layer to guarantee starvation freedom and to guard against known conflicts, and an inner protocol to guarantee mutual exclusion.

The inner protocol is the competition for *CS*. It uses process numbers for tie breaking, just as Lamport's Bakery algorithm [15]. We therefore represent the processes by natural numbers. If q and r are processes with $q < r$, we speak of q as the lower process and r as the higher process. For every pair of processes, the inner protocol gives priority to the lower process, and it lets the higher process determine compatibility. As processes can enter the inner protocol concurrently only within the margins allowed by the middle layer, we expect that the priority bias of the inner protocol is not very noticeable unless the load is so heavy that the performance of any algorithm would be problematic.

Despite the three layers, the central algorithm is rather simple. The outer protocol uses three messages for every process in *nbh*. The middle layer needs no additional messages. The inner protocol uses one message for every higher process in *nbh*, and no messages for the lower processes in *nbh*.

2.4 Into the code of the central algorithm

The code of the central algorithm is given in Figure 1. For an unbroken flow of control, we include the lines 22 and 23, which belong to the registration algorithm.

Every process p has a program counter $pc.p : \mathbb{N}$, which is initially 21. For a process p and a line number ℓ , we write p **at** ℓ to express $pc.p = \ell$. If L is a set of line numbers, we write p **in** L to express $pc.p \in L$.

Every process, say p , is in an infinite loop, in the line numbers 21 up to 28. It is at line 21 if and only if it is idle. Independently of its line number, it is always able and willing to receive messages from any other process, say q . In other words, in Figure 1, the eight alternatives of **central** are interleaved with the six alternatives of **receive**.

We regard the 14 alternatives of Figure 1 as atomic commands. This is allowed because actions on private variables give no interference, the messages are asynchronous, and any delay in sending a message can be regarded as a delay in message delivery.

One may regard the receiver as an independent thread of the looping process, with the guarantee that the 14 alternatives are executed atomically. Alternatively, an implementer may decide to schedule the receiver only when the process is idle (at line 21) or waiting at lines 22, 23, 24, 25, 26.

Every process has a private variable *job* of type *Job*, initially *none*. It has private variables *nbh*, *nbh0*, *prio*, *wack*, *after*, *away*, *need*, *prom*, which all hold finite sets of processes. All these sets are initially empty. It has the private variables *pcr*, *pack*, *fun*, and *curlist*, which serve in the registration algorithm and are treated in Section 2.7. The private variable *nbh0* is a history variable. It is set to *nbh* when the process executes line 25 and reset when the process leaves 28. It is not used in the algorithm, but serves in the proof of correctness.

Process p has a private extendable array *copy.p*, such that $copy.p(q) = job.q$ holds under suitable conditions. It is set when receiving **notify.q.p** and reset in **after**. We use the convention that $copy.p(q) = none$ when q is not in the current range of the array. Initially, the range of *copy.p* is empty.

At line 21, the environment gives process p a job. The lines 22 and 23 are treated in Section 2.7 with the registration algorithm.

For now we just assume that *nbh.p* gets some value before *curlist.p* becomes empty at line 23 and that, somehow, predicate *Rq1* is guaranteed.

The central algorithm uses four message keys: **notify**, **withdraw**, **ack**, **gra**. The messages **notify** hold values of the type *Job*, the other messages hold no values, they are of type *void*. The alternatives of **receive** with labels **after** and **prom** correspond to delayed answers. The message key **asklist** is treated in Section 2.7.

2.5 A layered solution

As announced, the central algorithm has three layers: an outer protocol to communicate the jobs, a middle layer to regulate access to the inner protocol, and an inner protocol for mutual exclusion. The three layers have waiting conditions in the lines 24, 25, 26, respectively. The outer protocol uses the messages **notify**, **withdraw**, **ack**, and the private variables *job*, *nbh*, *wack*, and *copy*. It can be obtained from Fig. 1 by removing the lines 22, 23, and all commands that use the messages **gra** and the private variables *prio*, *need*, *away*, *prom*. The middle layer consists of all commands that use *prio*. The inner protocol consists of the commands that use the messages **gra** and the private variables *need*, *away*, *prom*.

```

central( $p$ ) :
loop
21:   choose  $job \neq none$  ;
22:   await  $pc \leq 32$  ;
       $curlist := \{s \mid L(job)(s) > 0\}$  ;
      for all  $s \in curlist$  do  $asklist.p.s ! L(job)(s)$  od ;
23:   await  $curlist = \emptyset$  ; //  $nbh$  has been formed.
      for all  $q \in pack$  do  $hello.p.q !$  od ;
24:   await  $pack \cup wack = \emptyset$  ;
       $prio := \{q \mid q \notin after \wedge \neg job * copy(q)\}$  ;
25:   await  $prio = \emptyset$  ;  $nbh0 := nbh$  ;
      for all  $q \in nbh$  do  $notify.p.q ! job$  od ;
       $need := \{q \in nbh \mid p < q \vee (q \in away \wedge \neg job * copy(q))\}$  ;
26:   await  $need = \emptyset$  ;
27:    $CS$  ;
28:   for all  $q \in nbh$  do  $withdraw.p.q !$  od ;
       $wack := nbh$  ;  $job := none$  ;  $nbh := nbh0 := \emptyset$  ;
endloop .

receive( $p$ ) from( $q$ ) :
   $notify.q.p ? copy(q)$  ;
  if  $q < p$  then add  $q$  to  $prom$  endif .
   $withdraw.q.p ?$  ;
  add  $q$  to  $after$  ; remove  $q$  from  $prio$  ;
  if  $q < p$  then remove  $q$  from  $away$  and  $need$  endif .
   $after: \quad q \in after \wedge copy(q) \neq none \rightarrow$ 
     $ack.p.q !$  ; remove  $q$  from  $after$  ;  $copy(q) := none$  .
   $ack.q.p ?$  ; remove  $q$  from  $wack$  .
   $gra.q.p ?$  ; remove  $q$  from  $need$  .
   $prom: \quad q \in prom \wedge (pc \leq 26 \vee job * copy(q)) \rightarrow$ 
     $gra.p.q !$  ; add  $q$  to  $away$  ; remove  $q$  from  $prom$  ;
    if  $pc = 26 \wedge \neg job * copy(q)$ 
    then add  $q$  to  $need$  endif .
end receive .

```

Figure 1: The central algorithm for process p (with p 's private variables)

2.5.1 The outer protocol

In the outer protocol, every process p sends its job to all neighbours by means of **notify** messages in line 25, and it withdraws this in line 28. Reception of **notify** and **withdraw** is handled in the first three alternatives of **receive**. In the first line of **notify**, process p registers the job of q in $copy.p(q)$. The conditional statement of **notify** belongs to the inner protocol. When process p has received both **notify** and **withdraw** from q , it can execute the alternative **after** of **receive**, send **ack** back to q , and reset $copy.p(q) := none$. In this way, we allow the message **withdraw** to arrive before **notify**, even though it was sent later.

In the fourth alternative of **receive**, when process p receives an **ack** from q , it removes q from its set $wack$. This variable has been set by p to $nbh.p$ in line 28, while sending **withdraw** to its neighbours. When process p arrives again at line 24, it waits for $wack$ to be empty. In this way, it verifies that all its **withdraw** messages have been acknowledged, to preclude interference by delayed messages.

2.5.2 The inner protocol

The inner protocol serves to ensure the resource allocation condition $Rq2$. Every process forms in line 25 a set *need* of processes from which it needs “permission” to enter *CS*. As the condition q **in** *CS* is now equivalent to q **at** 27, condition $Rq2$ is implied by the invariants

$$\begin{aligned} Jq0: & \quad r \in need.q \Rightarrow q \text{ at } 26 \wedge r \in nbh0.q, \\ Rq2a: & \quad r \in nbh0.q \wedge q \in nbh0.r \\ & \quad \Rightarrow r \in need.q \vee q \in need.r \vee job.q * job.r. \end{aligned}$$

We postpone the proofs of these invariants to Section 3.5.

At this point, we break the symmetry. Recall that we represent the processes by natural numbers, and that, if $q < r$, we say that process q is *lower* and that r is *higher*. Notifications from lower processes are regarded as requests for permission that must be granted when possible, because we give priority to lower processes. Therefore, when process p receives **notify** from $q < p$, it stores q in *prom.p*. When the alternative **prom** is enabled, process p grants permission by sending **gra** to q . In *away.p*, it records the lower processes to which it has granted permission. If it is at line 26 and in conflict with q , it puts q in *need.p*.

There is a difference in the interpretation of *need.p* for lower and higher processes. If $q < p$, then $q \in need.p$ means that process p is in conflict with q and has granted priority to q . Process p therefore needs to wait for q ’s **withdraw** message. If $p < q$, then $q \in need.p$ means that process p has requested permission from q and is still waiting for the **gra** message (no conflict implied).

2.5.3 The middle layer

Without waiting at line 25, the algorithm of Figure 1 would satisfy $Rq2$, but it would have two defects. At line 26, one low process could repeatedly pass all higher conflicting neighbours. Also, long waiting queues of conflicting processes could form. These defects are treated by the middle layer.

When process p enters at line 24, it assigns to *prio.p* the set of processes with known incompatible *jobs*. This set is finite because it is contained in the finite set $\{q \mid copy.p(q) \neq none\}$. Process p then waits for the set *prio.p* to become empty. It removes q from *prio.p* when it receives **withdraw** from q . In this way, the middle layer only admits processes to the inner protocol that are not known to be conflicting with processes in the inner protocol. This improves the performance by making it unlikely that at line 26 long waiting queues of conflicting processes are formed. On the other hand, it ensures starvation freedom. In fact, when process p has executed line 25 and its **notify** messages have arrived, any conflicting neighbour of p that passes p at line 26, will have to wait for p at line 25, and hence cannot pass p again.

Remark. The first ideas for the present paper were tested in [11] in a context with a single resource. There, the **notify** messages are sent in the analogue of line 24 instead of line 25. This is also possible here. It has the effect that processes at line 25 are waiting for processes that arrived earlier at line 25. In other words, it induces a form of a first-come-first-served order. This is not a good idea for resource allocation. Consider, e.g., the following scenario.

Process p_0 arrives and starts using resource r_0 in *CS*. Now processes p_k for $k \geq 1$ arrive in their natural order at line 24 at intervals $> \Delta$ (see Section 1.4), needing the resources r_{k-1} and r_k , and with empty *wack*. If the notifications are sent in line 24, they all remain waiting at line 25, because $p_{k-1} \in prio.p_k$, until p_0 has passed *CS*. In the present version, with notifications sent at line 25, the processes with k odd start waiting at line 25, while the processes with k even go through to *CS*. \square

2.6 The job model

For the central algorithm, jobs are abstract objects with a compatibility relation. We need a job model for the registration algorithm.

In the simple case of exclusive access, one may regard every job as a set of resources, and define jobs to be compatible if and only if these sets are disjoint. In this case, we have $u * v \equiv (u \cap v = \emptyset)$ and $none = \emptyset$. If one wants to distinguish read requests from write requests, however, one needs a more complicated job model. In this case, one could model jobs as pairs of sets of resources, say (r, w) where r is the set of the resources for read access and w the set of resources for write access. Jobs (r_1, w_1) and (r_2, w_2) are then compatible if and only if the intersections $w_1 \cap w_2$, $w_1 \cap r_2$, and $r_1 \cap w_2$ are empty. One can also propose compatibility relations with more than two permission levels, where “shallow” access (e.g. reading of metadata) is allowed concurrently with “innocent” writing.

We therefore use a flexible job model that allows an arbitrary number $K \geq 1$ of levels. Let $upto(K)$ be the set $\{i \in \mathbb{N} \mid i \leq K\}$. Let Rsc be the set of resources. We characterize a job u as a function $Rsc \rightarrow upto(K)$, and define compatibility of jobs u and v by requiring that $u + v$ is at most K :

$$(0) \quad u * v \equiv (\forall c \in Rsc : u(c) + v(c) \leq K) .$$

In this way, relation $*$ is indeed symmetric, and the job $none$ given by $none(c) = 0$ for all c is compatible with all jobs.

The simple job model is the case with $K = 1$. We take $K = 2$ for the readers/writers problem. Read access at resource c requires $u(c) \geq 1$, write access requires $u(c) = 2$. In this way, concurrent reading is allowed, while writing needs exclusive access.

2.7 The registration algorithm

The registration algorithm serves to provide the processes with upperbound sets nbh . The idea is that every process registers for the resources that it needs, and that it then obtains lists of other registered clients. For the sake of flexibility, we assume that the resources are distributed over *sites* by means of a function $loc : Rsc \rightarrow Site$ from resources to sites. The sets Rsc and $Site$ are supposed to be finite. In order to preclude that they become bottlenecks, the sites get only the task to maintain a registration list and to answer to queries.

A process with a high registration level needs to perform much communication. We therefore offer the processes the option to lower their registration level, concurrently with the other activities.

We use the job model with upper bound K of Section 2.6. In particular, every job is a function $Rsc \rightarrow upto(K)$. A process can only use resource c at level k if it is registered at site $loc(c)$ for level $\geq k$. It therefore has an array fun such that $fun.p(s)$ is the p ’s registration level at site s . When some process obtains a new *job*, it needs at site s the level

$$L(job)(s) = \text{Max}\{job(c) \mid loc(c) = s\} .$$

For functions $f, g : Site \rightarrow \mathbb{N}$, we define $f \leq g$ to mean $(\forall s : f(s) \leq g(s))$.

In line 21 of Figure 1, the environment gives process p a new job. At line 22, process p may have to wait, to avoid interference with the lowering thread that is treated below. It then sends $L(job)(s)$ to site s if it is positive, and thus asks the site for a lists of clients that might compete for its resources. The set nbh gets its contents while the process waits at line 23, through messages from the sites in answer to `asklist`.

We give the sites very small tasks. Site s communicates with the processes by receiving messages **asklist** and **lower**, and answering by **answer** and **done**, respectively; this according to the code:

```

site( $s$ ) from( $q$ ) :
  || asklist. $q.s$  ?  $k$  ;  $list(q) := \max(list(q), k)$  ;
    answer. $s.q$  ! { $r \mid list(r) > K - k$ } .
  || lower. $q.s$  ?  $k$  ;  $list(q) := k$  ; done. $s.q$  ! .
end site .

```

In this code, $list$ is the private extendable array $list.s$ of s , that holds the levels of registered clients. The value 0 means not-registered. The answering messages **answer** contain the processes that are in potential conflict at the level k . If process q lowers at site s its level to k , it gets response **done** as an acknowledgment.

Process p receives the messages from site s according to:

```

listen( $p$ ) from( $s$ ) :
  || answer. $s.p$  ?  $v$  ;  $nbh := nbh \cup (v \setminus \{p\})$  ;
    if  $fun(s) < L(job)(s)$  then
       $pack := pack \cup (v \setminus \{p\})$  ;
       $fun(s) := L(job)(s)$  endif ;
       $curlist := curlist \setminus \{s\}$  .
  || done. $s.p$  ? ;  $reglist := reglist \setminus \{s\}$  .
end listen .

```

If process p increases its registration level at site s , it collects the potential competitors in the private variables $pack$. When it has received all answers, it sends all members of $pack$ a message **hello** in line 23, and waits for the responses **welcome** at line 24. The reason for this is that the members of $pack$ can be anywhere in their protocol and need not have $p \in nbh$.

The new messages **hello** and **welcome** are between processes. They are treated in the following two alternatives that should be included in **receive** of Figure 1.

```

|| hello. $q.p$  ? ;
  welcome. $p.q$  ! ( $pc \geq 26 \wedge q \notin nbh ? job : none$ ) ;
  if  $pc \geq 23$  then add  $q$  to  $nbh$  endif .
|| welcome. $q.p$  ?  $v$  ; remove  $q$  from  $pack$  ;
  if  $v \neq none$  then  $copy(q) := v$  endif .

```

If process p is in $\{26 \dots\}$ and $q \notin nbh.p$, the message **welcome** carries the job of q as a belated notification. Otherwise, it only holds *none* as an acknowledgement. Here, we use a conditional expression of the form $(b ? x : y)$ to mean x if b holds and otherwise y , as in the programming language C.

In **welcome**, the assignment to $copy.p(q)$ ensures that, when process p raises its registration level, it cannot enter its inner protocol when in conflict with q , while q remains in its inner protocol. At this point, the guard of line 25 is necessary for safety. This is a third reason for the middle layer of Section 2.5.3.

If process p receives **hello** and is in $\{23 \dots 25\}$, it adds q to nbh to notify it at line 25. If process p sends its job to q , it adds q to nbh because the job must be withdrawn later. At this point, the set $nbh0$ can become a proper subset of nbh .

Lowering means the choice of a new value $news$ for fun , which can be equal or lower than the current value. The processes can lower at the sites more or less concurrently with the loop 21–28. For this purpose, each of them gets a separate concurrent thread with a separate process counter pcr . We write q **at** ℓ to mean $pc.q = \ell$ if $\ell \in \{21 \dots 28\}$, and to mean $pcr.q = \ell$ if $\ell \in \{31 \dots 33\}$. If $\ell \in \{21 \dots 28\}$, we write q **in** $\{\ell \dots\}$ to mean $pc.q \geq \ell$. Initially, every process is both **at** 21 and **at** 31.

```

lowering( $p$ ) :
loop
31:   choose  $news$  with  $news \leq fun$  ;
32:   await  $pc = 21 \vee (pc \geq 25 \wedge L(job) \leq news)$  ;
       $reglist := \{s \mid news(s) \neq fun(s)\}$  ;  $fun := news$  ;
      for all  $s \in reglist$  do  $lower.p.s ! news(s)$  od ;
33:   await  $reglist = \emptyset$  ;
endloop .

```

The lowering thread shares the variable fun with the main thread, and has private variables $news$ and $reglist$. It is idle at line 31. When it is idle, to replace fun , the environment can give the process a value $news$. At line 32, the process informs the sites s for which the level is to be modified by sending them a message **lower** with the new value. If the main thread is in 22–24, the lowering thread needs to wait to avoid interference. The guard $L(job) \leq news$ is needed to protect the current job of p .

Initially, the private sets $pack.p$ and $reglist.p$ are empty and the functions $fun.p$, $news.p$, and $list.s$ are constant zero.

2.8 Aborting the entry protocol

As announced we give the environment the option to abort the entry protocol under certain circumstances:

```

abort( $p$ ) :
  |  $pc = 24 \wedge pack = \emptyset \rightarrow job := none ; nbh := \emptyset ; pc := 21$  .
  |  $pc = 25 \rightarrow job := none ; nbh := \emptyset ; prio := \emptyset ; pc := 21$  .
  |  $pc = 26 \wedge need \cap \{q \mid p < q\} = \emptyset \rightarrow$ 
    for all  $q \in nbh$  do  $withdraw.p.q !$  od ;  $wack := nbh$  ;
     $need := \emptyset ; job := none ; nbh := nbh0 := \emptyset ; pc := 21$  .

```

When the environment of p wants to abort the entry protocol at line 24, it may have to wait for emptiness of $pack$. This waiting is necessary to catch the expected **welcome** messages. It is shorter than 2Δ (see Section 1.4). At line 26, it may have to wait for emptiness of the higher part of $need$, necessary to catch the expected **gra** messages. This waiting is short because process p has priority over its higher neighbours. Indeed, the higher part of $need$ is empty after $\Gamma + 2\Delta$ (see Section 1.4).

To summarize, every process p has four concurrent threads that execute their atomic steps in an interleaved way. These threads are: the *environment* with the prompting steps at lines 21 and 31 and the 3 aborting steps, the *forward* thread with the steps at the lines 22–28; the *lowering* thread with the steps at the lines 32–33; and the *triggered* thread for the 6 messages from other processes, the 2 messages from sites, and the delayed answers **after** and **prom**. Moreover, every site has two commands for the message keys **asklist** and **lower**.

3 Verification of Safety

In this section, we prove the safety properties of the algorithm. For this purpose, we model the algorithm as a transition system that is amenable to formal verification.

The modelling, in particular of the asynchronous messages, is discussed in Section 3.1. In Section 3.2, we describe the verification of safety by means of invariants. In Section 3.3, we describe some choices we made to ease our proof management.

We first treat the central algorithm of Figure 1 and prove that it satisfies its requirement $Rq2$. In Section 3.4 we develop the invariants of the outer protocol

that are needed in the proof of $Rq2$. As indicated in Section 2.5.2, the mutual exclusion predicate $Rq2$ is implied by $Jq0$ and $Rq2a$. These invariants of the inner protocol are proved in Section 3.5, together with a number of auxiliary ones.

In Section 3.6, we add the registration algorithm, verify that the new messages are modelled correctly and that this addition does not disturb the safety properties of the central algorithm. Section 3.7 shows that it indeed serves its purpose and guarantees the invariant $Rq1$.

In Section 3.8, we conclude the list of invariants by presenting some invariants that are needed in the proofs of progress in the Sections 4 and 5.

3.1 Modelling of messages

We model the algorithm as a transition system with as state space the Cartesian product of the private state spaces augmented with the collection of messages in transit. The messages in transit are modelled by shared variables.

There are two kinds of messages: messages of type **void** without content but consisting of a single key word, and messages with content.

For a message key m of type **void**, we use $m.q.r$ as an integer shared variable that holds the number of messages m in transit from q to r , to be inspected and modified only by the processes q and r .

A sending command $m.p.q!$ from p to q e.g., as used in Figure 1, corresponds to an incrementation of $m.p.q$ by one, which can be denoted $m.p.q++$. A receiving command $m.q.p?$ from q by p followed by a statementlist S corresponds to a guarded alternative in which $m.q.p$ is decremented by one when positive:

$$\parallel \quad m.q.p > 0 \rightarrow m.q.p-- ; S .$$

The value of $m.q.r$ can be any natural number, but in our algorithm, we preserve the invariants $m.q.r \leq 1$: there is always at most one message m in transit from q to r . Initially, no messages are in transit: $m.q.r = 0$ for every message key m .

For messages with content, like **notify**, the above way of modelling cannot be used. In principle, we should model such messages by means of bags (multisets) of messages from sender to destination. In the algorithm, however, there is never more than one message with key m in transit from q to r . For simplicity, therefore, we model such a channel with key m from q to r as a shared variable $m.q.r$, which equals \perp if and only if there is no message m in transit from q to r .

In particular, we model the sending command **notify.p.q!** *job* from p to q at line 25 therefore as **notify.p.q** := *job*. Reception of **notify** from q by p followed by S is modelled by

$$\parallel \quad \text{notify.q.p} \neq \perp \rightarrow \\ \text{copy}(q) := \text{notify.q.p} ; \text{notify.q.p} := \perp ; S .$$

Initially, **notify.q.r** = \perp for all q and r .

As we model the bag by a single variable, we need to make sure that the bag has never more than one element. In other words, this way of modelling gives us the proof obligation that **notify.q.r** is sent only under the precondition **notify.q.r** = \perp . This will follow from the invariant $Iq7a$ of Section 3.4 below.

3.2 Using invariants

In a distributed algorithm, at any moment, many processes are able to do a step that modifies the global state of the system. The only way to reason successfully and reliably about such a system is to analyse the properties that cannot be falsified by any step of the system. These are the invariants.

Formally, a predicate is called an *invariant* of an algorithm if it holds in all reachable states. A predicate J is called *inductive* if it holds initially and every step of the algorithm from a state that satisfies J results in a state that also satisfies J . Every inductive predicate is an invariant. Every predicate implied by an invariant is an invariant.

When a predicate is inductive, this is often easily verified. In many cases, the proof assistant PVS is able to do it without user intervention. It always requires a big case distinction, because the transition system has many different alternatives.

Most invariants, however, are not inductive. Preservation of such a predicate by some alternatives needs the validity of other invariants in the precondition. We use PVS to pin down the problematic alternatives, but human intelligence is needed to determine the useful other invariants.

In proofs of invariants, we therefore use the phrase “*preservation of J at $\ell_1 \dots \ell_m$ follows from $J_1 \dots J_n$* ” to express that every step of the algorithm with precondition $J \wedge J_1 \dots J_n$ has the postcondition J , and that the additional predicates $J_1 \dots J_n$ are only needed for the alternatives $\ell_1 \dots \ell_m$.

We use the following names for the alternatives. The first 8 alternatives of **central** in Figure 1 are indicated by the line numbers. The alternatives of **receive** are indicated by the message names and the labels **after** and **prom**. We indicate the aborting alternatives of Section 2.8 by **ab24**, **ab25**, **ab26**.

For all invariants postulated, the easy proof that they hold initially is left to the reader. We use the term invariant in a premature way. See the end of this section.

3.3 Proof engineering

Effective management of the combined design and verification of such an algorithm requires a number of measures. We give most invariants names of the form Xqd , where X stands for an upper case letter and d for a digit. This enables us to reorder and rename the invariants in the text and the PVS proof files and to keep them consistent. Indeed, any modification of proof files must be done very carefully to avoid that the proof is destroyed. Using short distinctive names also makes it easy to search for definitions and to see when all of them have been treated.

Line numbers may change during design. In order to use query-replace for this in all documents, we use line numbers of two digits. In this way, we preclude that the invariants get renamed by accident. This is also the reason to use disjoint ranges for the line numbers of Figure 1 and the lowering thread of Section 2.7.

There is a trade off in the size of the invariants. Smaller invariants are easier to prove and easier to apply, but one needs more of them, and they are more difficult to remember. We therefore often combine a number of simple properties in a single invariant, see *Iq1* below. Bigger invariants are sometimes needed to express different aspects of a complicated state of affairs, compare *Iq2* below.

3.4 Invariants of the outer protocol

For now, we restrict ourselves to the transition system with 14 transitions of Figure 1. The nine steps of Section 2.7 are added in Sections 3.6 and 3.7. The three steps of Section 2.8 are added in Section 3.8.

We have two invariants about neighbourhoods:

$$\begin{aligned} Iq0: & \quad q \notin nbh.q, \\ Iq1: & \quad r \in nbh0.q \Rightarrow q \text{ in } \{26 \dots\} \wedge r \in nbh.q. \end{aligned}$$

These predicates are easily seen to be inductive.

At line 24, the processes wait for acknowledgements as expressed by emptiness of *wack*. This corresponds to the invariant:

$$Iq2: \quad \text{withdraw}.q.r + |q \in \text{after}.r| + \text{ack}.r.q = |r \in \text{wack}.q| .$$

Recall that $\text{withdraw}.q.r$ is the number of **withdraw** messages from q to r and that $\text{ack}.r.q$ is the number of **ack** messages from r to q . For Boolean b , we define $|b| \in \mathbb{N}$ to be 1 if b holds, and 0 otherwise. Predicate $Iq2$ is a concise expression of a complicated fact. Namely, $r \in \text{wack}.q$ holds if and only if there is a **withdraw** message in transit from q to r , or an **ack** message in transit from r to q , or $q \in \text{after}.r$. Furthermore, the three possibilities are mutually exclusive. Finally, there is at most one **withdraw** message from q to r , and at most one **ack** message from r to q . One could therefore split $Iq2$ into 9 different invariants.

Preservation of $Iq2$ when **withdraw** is sent at line 28 follows from the inductive invariant:

$$Iq3: \quad q \text{ in } \{25 \dots\} \Rightarrow \text{wack}.q = \emptyset .$$

For practical purposes, it is useful to notice that $Iq2$ and $Iq3$ together imply

$$Iq2a: \quad q \text{ in } \{25 \dots\} \Rightarrow \text{withdraw}.q.r = 0 \wedge q \notin \text{after}.r .$$

As announced, one of the functions of the outer protocol is to guarantee that, under suitable conditions, process r has the job of q in its variable $\text{copy}.r(q)$. In fact, the conditions are that r is in $\text{nbh0}.q$ and that there is no message **notify** in transit from q to r , as expressed in the invariant

$$Iq4: \quad r \in \text{nbh0}.q \wedge \text{notify}.q.r = \perp \Rightarrow \text{copy}.r(q) = \text{job}.q .$$

Preservation of $Iq4$ at line 21 follows from $Iq1$. Preservation at **after** follows from $Iq1$ and $Iq2a$. Preservation at line 25 and **notify** follows from $Iq1$ and the new invariants:

$$Iq5: \quad \text{job}.q = \text{none} \equiv q \text{ at } 21 ,$$

$$Iq6: \quad q \text{ in } \{26 \dots\} \wedge \text{notify}.q.r \neq \perp \Rightarrow \text{notify}.q.r = \text{job}.q .$$

Predicate $Iq5$ is inductive. Preservation of $Iq6$ at line 25 follows from the new invariant

$$Iq7a: \quad q \text{ at } 25 \Rightarrow \text{notify}.q.r = \perp \wedge \text{copy}.r(q) = \text{none} .$$

Predicate $Iq7a$ is logically implied by $Iq2$, $Iq3$, and the new invariant:

$$Iq7: \quad (\text{notify}.q.r = \perp \wedge \text{copy}.r(q) = \text{none}) \vee (q \text{ in } \{26 \dots\} \wedge r \in \text{nbh}.q) \\ \vee \text{withdraw}.q.r > 0 \vee q \in \text{after}.r .$$

Preservation of $Iq7$ at **after** follows from the new invariant:

$$Iq8: \quad \text{notify}.q.r = \perp \vee \text{copy}.r(q) = \text{none} .$$

Preservation of $Iq8$ at line 25 follows from $Iq7a$.

This is not circular reasoning: the above argument shows that, if all predicates Iq^* hold in the precondition of any step, they also hold in the postcondition. Therefore, the conjunction of them is inductive, and each of them is an invariant.

3.5 The proof of mutual exclusion

In Section 2.5.2, we saw that the mutual exclusion predicate $Rq2$ is implied by $Jq0$ and $Rq2a$. In this section, we prove that these two predicates are invariants.

Preservation of predicate $Jq0$ at **prom** follows from $Iq4$, $Jq0$, and the new invariants

$$\begin{aligned}
Rq1a: & \quad q \text{ in } \{26 \dots\} \wedge r \text{ in } \{26 \dots\} \Rightarrow q = r \vee r \in nbh0.q \vee job.q * job.r , \\
Jq1: & \quad q \in prom.r \Rightarrow q < r , \\
Jq2: & \quad q < r \Rightarrow |\text{notify}.q.r \neq \perp| + |q \in prom.r| + \text{gra}.r.q = |r \in need.q| .
\end{aligned}$$

Predicate $Rq1a$ is a strengthening of $Rq1$ of Section 2.1 that is guaranteed by the registration algorithm. This is shown in Section 3.7. Predicate $Jq1$ is inductive. Preservation of $Jq2$ at 25 and **prom** follows from $Iq5$, $Jq0$ and $Jq1$. Note the similarity of $Jq2$ with $Iq2$.

Predicate $Rq2a$ of Section 2.5.2 is implied by $Iq0$, $Iq1$, $Iq2a$, and the new invariants:

$$\begin{aligned}
Jq3: & \quad q < r \wedge r \in nbh0.q \Rightarrow r \in need.q \vee q \in away.r , \\
Jq4: & \quad q \in away.r \wedge q \in nbh0.r \wedge \text{withdraw}.q.r = 0 \\
& \quad \Rightarrow q \in need.r \vee job.q * job.r .
\end{aligned}$$

Preservation of $Jq3$ at **withdraw** follows from $Iq1$, $Iq2a$. At **gra**, it follows from the new invariant

$$Jq5: \quad \text{gra}.r.q > 0 \Rightarrow q \in away.r .$$

Preservation of $Jq4$ at 21 follows from $Iq1$. Preservation at **prom** follows from $Iq1$, $Iq4$, $Jq0$, $Jq1$, and $Jq2$. Preservation at line 25 and at **gra** and **withdraw** follows from $Iq4$, $Jq5$ and the new invariants:

$$\begin{aligned}
Jq6: & \quad q \in away.r \Rightarrow q < r \wedge \text{notify}.q.r = \perp , \\
Jq7: & \quad q \in away.r \wedge \text{withdraw}.q.r = 0 \Rightarrow r \in nbh0.q .
\end{aligned}$$

Preservation of $Jq5$ at **withdraw** follows from $Iq2a$, $Jq0$, and $Jq2$. Preservation of $Jq6$ follows at line 25 from $Iq1$, $Iq2$, $Iq3$, and $Jq7$, and at **prom** from $Jq1$, $Jq2$. Preservation of $Jq7$ at 25 and 28 follows from $Iq1$, and at **prom** and **withdraw** from $Jq0$, $Jq1$, $Jq2$, and $Jq6$.

This concludes the proof of the invariants $Jq0$ and $Rq2a$ under assumption of $Rq1a$.

3.6 Adding registration

We now add the registration algorithm to the central algorithm, i.e., we extend the transition system with the nine transitions of Section 2.7. There are three things to verify. The modeling must be correct, the proof of the central algorithm must not be disturbed, and condition $Rq1a$ of Section 3.5 must be guaranteed. The first two points are treated in this section. The third point is postponed to the next section.

The new messages **asklist**, **answer**, **welcome**, and **lower** are not void, and are therefore modelled in the same way as **notify**. This gives us the obligation to prove, for each of these four message keys m , that the value of m is \perp whenever a message m is sent. This follows from the invariants (similar to $Iq2$):

$$\begin{aligned}
Kq0: & \quad |\text{asklist}.q.s \neq \perp| + |\text{answer}.s.q \neq \perp| = |s \in curlist.q| , \\
Kq1: & \quad \text{hello}.q.r + |\text{welcome}.r.q \neq \perp| = |q \text{ at } 24 \wedge r \in pack.q| , \\
Kq2: & \quad |\text{lower}.q.s \neq \perp| + \text{done}.s.q = |q \text{ at } 33 \wedge s \in reglist.q| .
\end{aligned}$$

Predicate $Kq0$ is preserved at 22 because of the inductive invariant:

$$Kq3: \quad q \text{ at } 23 \vee curlist.q = \emptyset .$$

The invariants $Kq0$ and $Kq3$ together imply

$$Kq0a: \quad \text{answer}.s.q \neq \perp \Rightarrow q \text{ at } 23 .$$

Predicate $Kq1$ is preserved at **answer** because of $Kq0a$. Predicate $Kq2$ is inductive.

We turn to the question whether the central algorithm is disturbed by the new registration commands. The only variables that the registration algorithm shares with the central algorithm are pc , pcr , nbh , $pack$, and $copy$. Sharing pc is harmless, because the flow of control is not modified. Sharing $pack$ and pcr is harmless because the central algorithm has no invariants for $pack$ and pcr . Sharing nbh is almost harmless because all invariants except $Iq0$ allow enlarging nbh . Predicate $Iq0$ is preserved by **hello** because of $\text{hello}.q.q = 0$ which follows from $Kq1$ together with the inductive invariant

$$Kq4: \quad q \notin \text{pack}.q .$$

Modification of $copy$ by **welcome** requires new invariants. Predicate $Iq4$ is preserved at **welcome** because of $Iq1$ and the new invariant

$$Kq5: \quad q \text{ in } \{26 \dots\} \Rightarrow \text{welcome}.q.r \in \{\perp, \text{none}, \text{job}.q\} .$$

Predicate $Iq7$ is preserved at **welcome** because of

$$Kq6: \quad \text{welcome}.q.r \in \{\perp, \text{none}\} \vee \text{withdraw}.q.r > 0 \\ \vee q \in \text{after}.r \vee (q \text{ in } \{26 \dots\} \wedge r \in \text{nbh}.q) .$$

Predicate $Iq8$ is preserved at **welcome** because of

$$Kq7: \quad \text{welcome}.q.r \in \{\perp, \text{none}\} \vee (\text{notify}.q.r = \perp \wedge \text{copy}.r(q) = \text{none}) .$$

Predicate $Kq5$ is preserved at 25 because of $Iq2a$ and $Kq6$. Predicate $Kq6$ is preserved at **after** and **hello** because of $Iq1$ and $Kq7$. Predicate $Kq7$ is preserved at 25 and **hello** because of $Iq2a$, $Iq7$, and $Kq6$.

This concludes the proof that the central algorithm in combination with the registration algorithm preserves the invariant $Rq2$ of Section 2.1.

3.7 Safety of registration

We approach predicate $Rq1a$ in a bottom-up fashion. The lowering thread does not interfere with the main thread because of the inductive invariants:

$$Lq0: \quad q \text{ in } \{23, 24\} \Rightarrow q \text{ in } \{31, 32\} , \\ Lq1: \quad \text{news}.q \leq \text{fun}.q .$$

The sets $prio$ and $pack$ are only nonempty in specific locations, as expressed by the invariants:

$$Lq2: \quad q \in \text{prio}.r \Rightarrow r \text{ at } 25 \wedge q \notin \text{after}.r , \\ Lq3: \quad q \text{ in } \{23, 24\} \vee \text{pack}.q = \emptyset .$$

Indeed, $Lq2$ is inductive. Predicate $Lq3$ is preserved by **answer** because of $Kq0a$.

For the communication with the sites, we claim the invariants:

$$Lq4: \quad \text{asklist}.q.s \in \{\perp, L(\text{job}.q)(s)\} , \\ Lq5: \quad \text{lower}.q.s \in \{\perp, \text{fun}.q(s)\} .$$

Predicate $Lq4$ is preserved at 21 and 28 because $\text{asklist}.q.s \neq \perp$ implies $q \text{ at } 23$, as follows from $Kq0$ and $Kq3$. Predicate $Lq5$ is preserved at **answer** because of $Kq0a$, $Kq2$, and the mutual exclusion invariant $Lq0$.

There are subtle relations between $L(\text{job}.q)(s)$, $\text{fun}.q(s)$, and $\text{list}.s(q)$ expressed by the invariants:

$$Lq6: \quad q \text{ at } 22 \vee s \in \text{curlist}.q \vee L(\text{job}.q)(s) \leq \text{fun}.q(s) , \\ Lq7: \quad q \text{ in } \{23 \dots\} \wedge \text{asklist}.s.q = \perp \Rightarrow L(\text{job}.q)(s) \leq \text{list}.s(q) , \\ Lq8: \quad \text{fun}.q(s) \leq \text{list}.s(q) .$$

Predicate $Lq6$ is preserved at line 32 because of $Iq5$. Predicate $Lq7$ is preserved by **asklist** because of $Lq4$. It is preserved by **lower** because of $Kq2$, $Kq3$, $Lq0$, $Lq5$, and $Lq6$. Predicate $Lq8$ is preserved at 32 because of $Lq1$, at **answer** because of $Kq0$, $Kq3$, $Lq7$, and at **lower** because of $Lq5$.

After this preparation, we turn to the proof that the registration algorithm satisfies its purpose, i.e., guarantees predicate $Rq1a$ of Section 3.5. At line 23, process q expects and receives an answer from site s . This answer is a set that contains process r iff $L(job.q)(s) + list.s(r) > K$, where $list.s$ is the value at the time of sending the answer. In this way, we arrive at the invariant:

$$\begin{aligned} Mq0: \quad & q \text{ in } \{23\dots\} \Rightarrow q = r \vee r \in nbh.q \vee \text{hello}.r.q > 0 \\ & \vee (r \text{ at } 23 \wedge q \in \text{pack}.r) \vee L(job.q)(s) + \text{fun}.r(s) \leq K \\ & \vee (s \in \text{curlist}.q \wedge (\text{answer}.s.q = \perp \vee r \in \text{answer}.s.q)) . \end{aligned}$$

Predicate $Mq0$ is preserved at line 22, 32, **welcome**, **asklist** because of $Kq0a$, $Lq1$, $Kq1$, $Lq4$. It is preserved at **answer** because of $Kq0a$ and the new invariant:

$$\begin{aligned} Mq1: \quad & q \text{ in } \{23\dots\} \wedge \text{answer}.s.r \neq \perp \\ & \Rightarrow q = r \vee r \in nbh.q \vee q \in \text{answer}.s.r \\ & \vee L(job.q)(s) + L(job.r)(s) \leq K \\ & \vee (s \in \text{curlist}.q \wedge (\text{answer}.s.q = \perp \vee r \in \text{answer}.s.q)) . \end{aligned}$$

Predicate $Mq1$ is preserved at 21, 22, 28 because of $Kq0a$. It is preserved by **asklist** because of $Kq0$, $Kq3$, $Lq4$, $Lq7$.

The predicates $Mq0$, $Kq1$, $Kq3$, $Lq6$ together imply the derived invariant:

$$\begin{aligned} Mq0a: \quad & q \text{ in } \{24\dots\} \wedge r \text{ in } \{24\dots\} \wedge \text{pack}.r = \emptyset \\ & \Rightarrow q = r \vee r \in nbh.q \vee job.q * job.r . \end{aligned}$$

Predicate $Mq0a$ approximates $Rq1a$, but the disjunct $r \in nbh.q$ of $Mq0a$ is weaker than the disjunct $r \in nbh0.q$ of $Rq1a$. As a remedy, we postulate the invariant:

$$\begin{aligned} Mq2: \quad & q \text{ in } \{26\dots\} \wedge r \in nbh.q \\ & \Rightarrow r \in nbh0.q \vee \text{welcome}.q.r = job.q \vee \text{copy}.r(q) = job.q . \end{aligned}$$

Predicate $Mq2$ is preserved at **notify**, **answer**, **after**, **welcome**, **hello** because of $Lq1$, $Iq2a$, $Iq5$, $Iq8$, $Kq0a$, $Kq1$, $Kq5$.

The conjunction of $Mq0a$ and $Mq2$ is not strong enough to imply $Rq1a$. We need yet another invariant. Indeed, predicate $Rq1a$ is implied by $Lq2$ and the new invariant

$$\begin{aligned} Mq3: \quad & q \text{ in } \{26\dots\} \wedge r \text{ in } \{25\dots\} \\ & \Rightarrow q = r \vee r \in nbh0.q \vee q \in \text{prio}.r \vee job.q * job.r . \end{aligned}$$

Predicate $Lq2$ serves to eliminate the alternative $q \in \text{prio}.r$ of $Mq3$. Here we see, as announced at the end of Section 2.7, that correctness of the registration algorithm relies on the guard of line 25, the emptiness of prio .

Predicate $Mq3$ is preserved at lines 24 and 25 because of $Mq0a$ and $Iq2a$, $Iq5$, $Kq1$, $Lq3$, $Mq2$. It is preserved at **welcome** because of $Iq2a$.

This concludes the proof of $Rq1a$ of Section 3.5 and thus of $Rq1$ and $Rq0$ of Section 2.1.

3.8 Invariants concluded

We conclude the section by developing some invariants needed for the proofs of progress in the Sections 4 and 5. Primarily, we need additional invariants about *need*, *copy*, *prio*, because of the guards of the alternatives 25, 26, **after**, and **prom**.

We need two invariants for *need* in the inner protocol:

$Nq0: \quad q < r \wedge q \in need.r \Rightarrow q \in away.r$,
 $Nq1: \quad q < r \wedge q \in need.r \wedge job.q * job.r \Rightarrow withdraw.q.r > 0$.

Predicate $Nq0$ is inductive. Predicate $Nq1$ is preserved at 21 because of $Iq5$, at 25 because of $Iq1$, $Iq4$, $Jq6$ and $Jq7$, and at 28 because of $Iq1$, $Jq0$, $Jq7$, and $Nq0$. It is preserved at **prom** because of $Iq4$, $Jq0$, and $Jq2$.

We also need a new invariant of the outer protocol:

$Nq2: \quad notify.q.r = \perp \wedge copy.r(q) = none \wedge welcome.q.r \in \{\perp, none\}$
 $\Rightarrow q \notin after.r \wedge withdraw.q.r = 0$.

Preservation of $Nq2$ at 28, **hello**, **after**, and **notify** follows from $Iq1$, $Iq2$, $Iq4$, $Iq5$, $Kq1$, $Mq2$, and the new postulate

$Nq3: \quad notify.q.r \neq none$.

Predicate $Nq3$ is preserved at 25 because of $Iq5$.

Next to $Lq2$, we need a second invariant about *prio*:

$Nq4: \quad q \in prio.r \Rightarrow \neg copy.r(q) * job.r$.

Predicate $Nq4$ is preserved at 21, 28, and **after** because of $Lq2$. It is preserved at **notify** and **welcome** because of $Iq8$ and $Kq7$. This concludes the construction of the invariants for progress.

At this point, we incorporate the aborting commands of Section 2.8. These commands preserve all invariants claimed. In most cases, they need the same auxiliary invariants as line 28, because they share several actions with line 28, e.g., resetting *pc*, *job*, *nbh*, *nbh0*.

We now summarize the argument for safety by forming the conjunction of the universal quantifications of the predicates of the families Iq^* , Jq^* , Kq^* , Lq^* , Mq^* , Nq^* (the so-called constituent invariants). As verified mechanically, this conjunction is inductive. Such mechanical verification is relevant, because with more than 40 invariants, the possibility of overlooking an unjustified assumption or a clerical error is significant. As each of the constituent invariants is a consequence of the conjunction, each of them is itself invariant, as are all logical consequences of them. In particular, the mutual exclusion predicate $Rq0$ is invariant. This concludes the proof that the algorithm satisfies $Rq0$.

One may wonder whether the constituent invariants are independent. We do believe so, but we have no suitable way to verify this.

4 Progress

The algorithm satisfies strong progress properties. In this section, we introduce and formalize the progress properties.

We introduce weak fairness in Section 4.1. Section 4.2 contains the formalization in (linear-time) temporal logic. Section 4.3 formalizes weak fairness. In Section 4.4, we introduce absence of localized starvation to unify starvation freedom and concurrency as announced in Section 2.2, and we announce the main progress result: Theorem 1.

As a preparation for the proof of this result, we derive in Section 4.5 some invariants that relate disabledness to the occurrence of conflicts. These invariants are used in Section 4.6 to prove absence of localized deadlock. This result could be proved as a simple consequence of the main theorem. We prove it independently, because it nicely represents the main ideas of the proof of Theorem 1 in a simpler context.

4.1 Weak fairness

First, however, weak fairness needs an explanation. Roughly speaking, a system is called weakly fair if, whenever some process from some time onward always can do a step, it will do the step. Yet if a process is idle, it must not be forced to be interested in CS. Similarly, if a process is waiting a long time in the entry protocol, we do not want it to be forced to abort the entry protocol. We therefore do not enforce the environment to do steps. We thus exclude the environment from the weak fairness conditions.

Formally, we do not argue about the fairness of systems, but characterize the executions they can perform. Recall that an *execution* is an infinite sequence of states that starts in an initial state and for which every pair of subsequent states satisfies the next-state relation. The next-state relation is defined as the union of a number of step relations.

An execution is called *weakly fair* for a step relation iff, when the step relation is from some state onward always enabled, it will eventually be taken. For example, if some process p is at line 22, we expect that p will eventually execute line 22. If some message m from q to r is in transit, we expect that r eventually receives message m . By imposing weak fairness for some step relations, we restrict the attention to the executions that are weakly fair for these step relations.

We partition the steps of our algorithm in four classes (compare the end of Section 2.8). Firstly, we have the 5 *environment steps* 21, 31, **ab24**, **ab25**, **ab26**. Secondly, we have the 7 *forward steps* of the lines 22, 23, 24, 25, 26, 27, 28. The third class consists of the 12 *triggered steps*, the ten message reception steps of the processes and the sites and the two “delayed answers”: **after** and **prom**. The fourth class consists of the two *lowering steps* at the lines 32 and 33.

We distinguish the five environment steps because we don’t want to grant them weak fairness. A process that is idle for a long time must not be forced to get a job. A process that stays in its entry protocol must not be forced to abort it.

4.2 Formalization in temporal logic

Let X be the state space of the algorithm. This is the Cartesian product of the private state spaces of the processes and the sites, together with the sets of messages in transit. Executions are infinite sequences of states, i.e., elements of the set X^ω . For a state sequence $xs \in X^\omega$, we write $xs(n)$ for the n th element of xs . Occasionally, we refer to $xs(n)$ as the state at time n . For a programming variable v , we write $xs(n).v$ for the value of v in state $xs(n)$.

For a set of states $U \subseteq X$, we define $\llbracket U \rrbracket \subseteq X^\omega$ as the set of infinite sequences xs with $xs(0) \in U$. For a relation $A \subseteq X^2$, we define $\llbracket A \rrbracket_2 \subseteq X^\omega$ as the set of sequences xs with $(xs(0), xs(1)) \in A$.

For $xs \in X^\omega$ and $k \in \mathbb{N}$, we define the sequence $drop(k, xs)$ by $drop(k, xs)(n) = xs(k + n)$. For a subset $P \subseteq X^\omega$ we define $\Box P$ (*always* P) and $\Diamond P$ (*eventually* P) as the subsets of X^ω given by

$$\begin{aligned} xs \in \Box P &\equiv (\forall k \in \mathbb{N} : drop(k, xs) \in P) , \\ xs \in \Diamond P &\equiv (\exists k \in \mathbb{N} : drop(k, xs) \in P) . \end{aligned}$$

We now apply this to the algorithm. We write $init \subseteq X$ for the set of initial states and $step \subseteq X^2$ for the next state relation. Following [1], we use the convention that relation $step$ is reflexive (contains the identity relation). An *execution* is an infinite sequence of states that starts in an initial state and in which each subsequent pair of states is connected by a step. The set of executions of the algorithm is therefore

$$Ex = \llbracket init \rrbracket \cap \Box \llbracket step \rrbracket_2 .$$

If J is an invariant of the system, it holds in all states of every execution. We therefore have $Ex \subseteq \Box J$.

We define $(q \text{ at } \ell)$ to be the subset of X of the states in which process q is at line ℓ . An execution in which process q is always eventually at line 21, is therefore an element of $\Box \Diamond [q \text{ at } 21]$.

Remark. Note the difference between $\Box \Diamond [U]$ and $\Diamond \Box [U]$. In general, $\Box \Diamond [U]$ is a bigger set (a weaker condition) than $\Diamond \Box [U]$. The first set contains all sequences that are infinitely often in U , the second set contains the sequences that are from some time onward eternally in U . \square

4.3 Weak fairness formalized

For a relation $R \subseteq X^2$, we define the *disabled* set $D(R) = \{x \mid \forall y : (x, y) \notin R\}$. Now *weak fairness* [16] for R is defined as the set of executions in which R is infinitely often disabled or taken:

$$wf(R) = Ex \cap \Box \Diamond ([D(R)] \cup [R]_2) .$$

For our algorithm, the next state relation $step \subseteq X^2$ is the union of the identity relation on X (because $step$ should be reflexive) with the relations $step(p)$ that consists of the state pairs (x, y) where y is a state obtained when process p does a step starting in x . In accordance with Section 4.1, the steps that process p can do are partitioned as:

$$step(p) = env(p) \cup fwd(p) \cup trig(p) \cup low(p) ,$$

where $env(p)$, $fwd(p)$, $trig(p)$, $low(p)$ consist of the environment steps, the forward steps, the triggered steps, and the lowering steps of p , respectively. The set of triggered steps of p is a union:

$$trig(p) = \bigcup_{q,m} rec(m, q, p) \cup \bigcup_{s,n} sit(n, p, s) ,$$

where $rec(m, q, p)$ consists of the steps where p receives message m from q . Note that we take the union here over all processes q and the eight message alternatives m , including the delayed answers **after** and **prom**, and $sit(n, p, s)$ consists of the four commands for message keys n between process p and site s .

The set $wf(fwd(p))$ consists of the executions for which every forward step of process p is infinitely often disabled or taken.

The set $wf(rec(m, q, p))$ consists of the executions for which every message m in transit from q to p is eventually received.

An execution is defined to be *weakly fair for* process p if it is weakly fair for the forward steps of p , for the lowering steps of p , and for all messages with p as destination or source, as captured in the definition

$$Wf(p) = wf(fwd(p)) \cap wf(low(p)) \cap \bigcap_{s,n} wf(sit(n, p, s)) \\ \cap \bigcap_{q,m} (wf(rec(m, q, p)) \cap wf(rec(m, p, q))) ,$$

where s ranges over the sites, n over the messages to and from sites, q over the processes, and m over the eight message alternatives.

4.4 Absence of localized starvation

As discussed in Section 2.2, there are two progress properties to consider: starvation freedom, which means that every process that needs to enter *CS* will eventually do so unless its entry protocol is aborted, and concurrency, which means that every process that needs to enter *CS* and does not abort its entry protocol will eventually enter *CS* unless it comes in eternal conflict with some other process.

For starvation freedom we assume weak fairness for the forward and lowering steps and all triggered steps of all processes. For concurrency for process p , we only need weak fairness for the forward and lowering steps of process p itself and for all triggered steps in which process p is involved. As we want to verify both properties with a single proof, we unify them in the concept of absence of localized starvation. The starting point for the unification is that the concepts differ in the sets of steps that satisfy weak fairness.

Let W be a nonempty set of processes. *Absence of W -starvation* is defined to mean that weak fairness for all forward and lowering steps of the processes in W and for all triggered steps that involve processes in W implies that every process in W eventually comes back to line 21, unless it comes in eternal conflict with some process outside W . The special case that W is the set of all processes is starvation freedom. The special case that W is a singleton set is concurrency [4, 22].

We speak of *absence of localized starvation* if absence of W -starvation holds for every nonempty set W . The aim is thus to prove that the algorithm satisfies absence of localized starvation. In order to do so, we formalize the definition in terms of temporal logic.

We define the set of states where processes q and r are in conflict as $q \bowtie r$ by

$$q \bowtie r \equiv \neg \text{job}.q * \text{job}.r .$$

An execution where q is eventually in eternal conflict with r is therefore an element of $\Diamond \Box [q \bowtie r]$. Absence of W -starvation thus means that all “sufficiently fair” executions are elements of the set

$$(\bigcap_{q \in W} \Box \Diamond [q \text{ at } 21]) \cup (\bigcup_{q \in W, r \notin W} \Diamond \Box [q \bowtie r]) .$$

An execution is defined to be weakly fair for a nonempty set of processes W if it is weakly fair for each of them:

$$Wf(W) = \bigcap_{p \in W} Wf(p) .$$

Absence of localized starvation thus is the following result:

Theorem 1 $Wf(W) \subseteq (\bigcap_{q \in W} \Box \Diamond [q \text{ at } 21]) \cup (\bigcup_{q \in W, r \notin W} \Diamond \Box [q \bowtie r])$ holds for every nonempty set W of processes.

The proof of the Theorem is given in Section 5.

4.5 Disabledness and conflicts

As a preparation of the proof of Theorem 1, we use the invariants obtained in Section 3 to derive four so-called *waiting invariants* that focus on disabledness of processes in relation to the occurrence of conflicts. Forward steps can be disabled at the lines 23, 24, 25, 26 by nonemptiness of *curlist*, *wack*, *prio*, *need*, respectively. Message reception is disabled when there is no message.

Let $dAfter(q, r)$ and $dProm(q, r)$ be the conditions, respectively, that the alternatives **after** and **prom** for sending **ack** or **gra** from q to r are disabled:

$$\begin{aligned} dAfter(q, r) &\equiv r \notin \text{after}.q \vee \text{copy}.q(r) = \text{none} , \\ dProm(q, r) &\equiv r \notin \text{prom}.q \\ &\quad \vee (q \text{ in } \{27 \dots\} \wedge \neg \text{job}.q * \text{copy}.q(r)) . \end{aligned}$$

For emptiness of *wack*. q , the invariants $Iq2$ and $Nq2$ imply the waiting invariant:

$$\begin{aligned} Waq0: \quad &\text{withdraw}.q.r = \text{ack}.r.q = 0 \wedge \text{notify}.q.r = \perp \\ &\wedge \text{welcome}.q.r \in \{\perp, \text{none}\} \wedge dAfter(r, q) \Rightarrow r \notin \text{wack}.q . \end{aligned}$$

For emptiness of $prio.q$, the invariants $Kq7$, $Lq2$, $Mq2$, and $Nq4$, together with $Iq4$, $Iq5$, $Iq7$, and $Iq8$ imply the waiting invariant

$$\begin{aligned} \text{Waq1: } & r \in prio.q \wedge \text{withdraw}.r.q = 0 \wedge \text{welcome}.q.r \in \{\perp, none\} \\ & \Rightarrow r \text{ in } \{26 \dots\} \wedge q \bowtie r . \end{aligned}$$

For emptiness of $need.q$, we are forced to make a case distinction. Using the invariants $Nq0$ and $Nq1$ together with $Iq1$, $Jq0$, and $Jq7$, we obtain the waiting invariant

$$\text{Waq2: } r < q \wedge r \in need.q \wedge \text{withdraw}.r.q = 0 \Rightarrow r \text{ in } \{26 \dots\} \wedge q \bowtie r .$$

It follows from $Iq4$, $Iq5$, $Jq0$, and $Jq2$, that we have

$$\begin{aligned} \text{Waq3: } & q < r \wedge r \in need.q \wedge \text{gra}.r.q = 0 \wedge \text{notify}.q.r = \perp \\ & \wedge dProm(r, q) \Rightarrow r \text{ in } \{27 \dots\} \wedge q \bowtie r . \end{aligned}$$

4.6 Intermezzo: absence of localized deadlock

In this section we prove absence of localized deadlock. This result is not useful for the proof of Theorem 1, and it follows from Theorem 1. Yet, an independent proof of the result is a good preparation of the more complicated proof that follows.

Informally speaking, absence of localized deadlock means that, when none of the processes of some set W of processes can do a forward or lowering or triggered step, and some of them are not at line 21, then at least one of them is in conflict with a process not in W . It is thus a safety property.

The concept is defined as follows. We define a process p to be *silent* in some state when every forward or lowering or triggered step of p is disabled, and no process or site can do a triggered step that sends a message to p . We define p to be *locked* when it is silent and not at line 21.

Let W be a set of processes (willing to do steps). The set W is said to be *silent* if all its processes are silent. It is said to be *locked* if it is silent and contains locked processes. Absence of W -deadlock is the assertion that, if W is locked, then it contains some process that is in conflict with a process not in W . Absence of localized deadlock is absence of W -deadlock for every set W . This is our next result:

Theorem 2 *Assume that a set W of processes is locked. Then there are processes $q \in W$ and $r \notin W$ with $q \bowtie r$.*

Proof. The algorithm clearly satisfies the invariant that every process is always $\{21 \dots 28\}$. The processes in $\{27, 28\}$ can do a forward step. Therefore, all processes in W are in $\{21 \dots 26\}$. As W contains locked processes, there are processes in W at 22–26, waiting for $pcr \leq 32$ or emptiness of *curlist*, *wack*, *prio*, *need*, respectively. If $p \in W$ has $pc.p = 22$ then $pcr.p > 32$ and hence $pcr.p = 33$; therefore process p can do a lowering step. This implies that W has no processes at line 22.

As all triggered steps for processes in W are disabled, we have $\text{asklist}.p.s = \text{answer}.s.p = \perp$ for all $p \in W$. By $Kq0$, this implies $\text{curlist}.p = \emptyset$ for all $p \in W$. It follows that W has no processes at line 23.

Similarly, for all pairs q, r with $q \in W$ or $r \in W$, the values of $\text{notify}.q.r$ and $\text{welcome}.q.r$ are \perp and the values of $\text{hello}.q.r$, $\text{withdraw}.q.r$, $\text{ack}.q.r$, and $\text{gra}.q.r$ are all 0. Also, $dAfter(q, r)$ and $dProm(q, r)$ hold. This simplifies the waiting invariants Waq^* of Section 4.5 considerably. In fact, for $q \in W$ and r arbitrary, we obtain:

$$\begin{aligned} \text{Wax0: } & r \notin \text{wack}.q , \\ \text{Wax1: } & r \in prio.q \Rightarrow r \text{ in } \{26 \dots\} \wedge q \bowtie r , \\ \text{Wax2: } & r < q \wedge r \in need.q \Rightarrow r \text{ in } \{26 \dots\} \wedge q \bowtie r , \\ \text{Wax3: } & q < r \wedge r \in need.q \Rightarrow r \text{ in } \{27 \dots\} \wedge q \bowtie r . \end{aligned}$$

It follows from *Kq1* and *Wax0* that W has no processes disabled at 24.

Assume that W contains processes disabled at 26. Let q be the lowest process in W waiting at 26. Because q is disabled at 26, the set $need.q$ is nonempty, say $r \in need.q$. It follows from *Iq0* and *Jq0* that $r \neq q$. Then *Wax2* and *Wax3* imply $q \bowtie r$. Moreover, if $q < r$, then r is in $\{27 \dots\}$ by *Wax3*, so that $r \notin W$. On the other hand, if $r < q$, then r is in $\{26 \dots\}$ by *Wax2*. If $r \in W$, then r would be at 26, contradicting the minimality of q . This proves that $r \notin W$ in either case.

Assume that W contains no processes disabled at 26. Then it has some process $q \in W$ disabled at 25. Because q is disabled at 25, the set $prio.q$ is nonempty, say $r \in prio.q$. Now *Wax1* implies that r is in $\{26 \dots\}$ and $q \bowtie r$. Because r is in $\{26 \dots\}$, it is not in W . \square

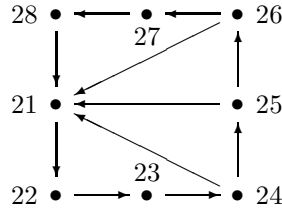
5 Verification of Progress

In this section, we prove progress of the algorithm. We prepare the proof of Theorem 1 by concentrating on what can be inferred from weak fairness for a single process.

Following [5], for sets of states U and V , we define U **unless** V to mean that every step of the algorithm with precondition $U \setminus V$ has the postcondition $U \cup V$. The algorithm has the obvious **unless** relations:

- (1) $(p \text{ at } \ell) \text{ unless } (p \text{ at } \{\ell + 1\})$ for $\ell \in \{22, 23, 27\}$,
 $(p \text{ at } \ell) \text{ unless } (p \text{ in } \{21, \ell + 1\})$ for $\ell \in \{24, 25, 26\}$,
 $(p \text{ at } 28) \text{ unless } (p \text{ at } 21)$.

The jumps from 24, 25, 26 back to 21 are due to the aborting commands of Section 2.8. In any case, all cycles go through line 21:



Weak fairness of process p itself suffices for progress at the lines 27 and 28, because the forward commands at these lines are always enabled. We thus have

- (2) $Wf(p) \cap \Diamond \Box [p \text{ at } \ell] = \emptyset$ for $\ell \in \{27, 28\}$.

We treat progress at the lines 22 up to 26 in the Sections 5.1 up to 5.5. The proof of Theorem 1 is concluded in Section 5.6. Progress of the lowering thread is proved in Section 5.7.

5.1 Progress at line 22

In order to prove progress at line 22, in view of the guard $pcr \leq 32$, we first need to prove progress of the lowering thread at line 33.

Assume that the lowering thread of process p remains eternally at line 33, say from time n_0 onward. From this time onward, the finite set $reglist.p$ is only modified by removing elements from it (by **done**). Therefore, $reglist.p$ becomes eventually constant. If it becomes eventually empty, the step of line 33 is eventually always enabled, so that process p will move to line 31, contradicting the assumption. Therefore, there is a site s that remains eternally in $reglist.p$. By the invariant *Kq2*, it therefore follows that, from time n_0 onward, we have $\text{lower}.p.s \neq \perp$ or $\text{done}.s.p > 0$. By weak fairness, any message $\text{lower}.p.s$ is eventually received, and never sent again. The same holds for $\text{done}.s.p$. This is a contradiction, thus proving that

$$(3) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 33] = \emptyset .$$

This kind of argument will be used again. We refer to it as a *shrinking argument*.

Now assume that process p remains eternally at line 22. Its lowering thread, if at line 33, will leave line 33, and never come back again because of the guard of line 32. Therefore process p is eventually always enabled at line 22. By weak fairness, it will eventually leave line 22, a contradiction. This proves

$$(4) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 22] = \emptyset .$$

5.2 Progress at line 23

For some process p , assume that xs is an execution in $\Diamond \Box [p \text{ at } 23]$, weakly fair for p . By the shrinking argument used above, the set $curlist.p$ is eventually constant. Using the invariants $Kq0$ and $Kq3$, and weak fairness for the messages **asklist** and **answer**, we get that $curlist.p$ is eventually empty. Then using weak fairness for its forward steps, we see that process p eventually leaves line 23, contradicting the assumption. This proves that

$$(5) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 23] = \emptyset .$$

5.3 Progress at line 24

For some process p , assume that xs is an execution in $\Diamond \Box [p \text{ at } 24]$, weakly fair for p . From some time n_0 onward, process p is and remains at line 24. By the shrinking argument, the sets $pack.p$ and $wack.p$ are eventually constant. By the invariant $Kq1$ and weak fairness of the messages **hello** and **welcome**, the set $pack.p$ is eventually empty. Using weak fairness of the forward steps of p , we get that $wack.p$ remains nonempty. This implies that there is a process q such that, from time n_0 onward, $q \in wack.p$ always holds.

We next note that, as process p is eventually always at line 24, we have eventually always $notify.p.q = \perp$ and $withdraw.p.q = 0$ and $welcome.p.q \in \{\perp, none\}$. In fact, once process p is and remains at line 24, it will not send any messages **notify** or **withdraw**, or any messages **welcome** $\neq none$. Weak fairness ensures that any such messages are received eventually. This proves that there is a time $n_1 \geq n_0$ such that, from time n_1 onward, we have $notify.p.q = \perp$ and $withdraw.p.q = 0$ and $welcome.p.q \in \{\perp, none\}$. Now $Waq0$ implies that $\neg dAfter(p, q) \vee ack.q.p > 0$ holds from time n_1 onward. By weak fairness, process q will send **ack** to p . By weak fairness, this **ack** will be received, and p will remove q from $wack.p$. This contradiction proves:

$$(6) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 24] = \emptyset .$$

5.4 Progress at line 25

For some process p , assume that xs is an execution in $\Diamond \Box [p \text{ at } 25]$, weakly fair for p . From some time n_0 onward, process p is and remains at line 25. By the shrinking argument, there is a process q such that, from time n_0 onward, $q \in prio.p$ always holds. If $withdraw.q.p > 0$ holds at some time $n \geq n_0$, by weak fairness this message will eventually arrive and falsify $q \in prio.p$. Therefore, $withdraw.q.p = 0$ holds from time n_0 onward. By the above argument, there is a time $n_1 \geq n_0$ such that $welcome.p.q \in \{\perp, none\}$ holds from time n_1 onward. By $Waq1$, q is in $\{26 \dots\}$ and $p \bowtie q$ holds from time n_1 onward. This proves

$$(7) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 25] \subseteq \bigcup_q \Diamond \Box [q \text{ in } \{26 \dots\} \wedge p \bowtie q] .$$

5.5 Progress at line 26

For some process p , let xs be an execution in $\Diamond\Box[p \text{ at } 26]$, weakly fair for p . Process p waits at line 26 for emptiness of $need$. This condition belongs to the inner protocol. The inner protocol in isolation, however, is not starvation free because it would allow a lower process repeatedly to claim priority over p by sending notifications. We need the outer protocol to preclude this. Technically, the problem is that $need.p$ can grow at line 26 because of the alternative **prom**.

We therefore investigate conditions under which the predicate $q \in need.p$ or its negation is stable, i.e., once true remains true. As p remains at line 26, the set $nbh0.p$ remains constant. For $q \notin nbh0.p$, we have $q \notin need.p$ by $Jq0$. For $q \in nbh0.p$, we define the predicate

$$bf(q, p) : \quad q \text{ in } \{\dots 24\} \vee job.p * job.q \vee p \in prio.q .$$

Roughly speaking, $bf(q, p)$ expresses that q is not in the inner protocol or is not in conflict with p . While process p is and remains at line 26 and $copy.q(p) \neq none$, the predicate $bf(q, p)$ is stable. The main point is when process q executes line 24. If the second disjunct of $bf(q, p)$ does not hold, process q puts p into $prio.q$. The proof uses $Iq2a$, $Iq4$, $Iq7$, $Iq8$, and $Rq1a$.

While process p is and remains at line 26 and either $bf(q, p)$ holds or $p < q$, the predicate $q \notin need.p$ is stable. This is proved at the alternative **prom** with $Iq4$, $Jq0$, $Jq1$, $Jq2$, and $Lq2$.

While process p is and remains at line 26 and $bf(q, p)$ is false and $q \leq p$, the predicate $q \in need.p$ is stable. This is proved at the alternatives **gra** and **withdraw** with $Iq2a$, $Jq5$.

We can now combine these predicates in the variant function

$$\begin{aligned} vf(q, p) &= (bf(q, p) ? \quad |q \in need.p| \\ &\quad : q \in need.p ? \quad 3 \\ &\quad : p < q ? \quad 2 : 4) . \end{aligned}$$

Here, we write $(B ? x : y)$ for the conditional expression that equals x when B holds, and otherwise y , as in the programming language C. Clearly, $vf(q, p)$ is odd (1 or 3) if and only if $q \in need.p$ holds. Similarly, $vf(q, p) \leq 1$ holds if and only if $bf(q, p)$.

The above stability results about $bf(q, p)$ and $q \in need.p$ and its negation imply that, while p is and remains at line 26 and $copy.q(p) \neq none$, the function $vf(q, p)$ never increases. By weak fairness, eventually $notify.p.q = \perp$ holds. As process p remains at line 26, $notify.p.q = \perp$ remains valid. The invariant $Iq4$ together with $Iq5$ and $Iq1$ therefore implies that $copy.q(p) \neq none$ holds and remains valid because $q \in nbh0.p$. Therefore, from that time onward, $vf(q, p)$ never increases.

It follows that, for every $q \in nbh0.p$, eventually, $vf(q, p)$ gets a constant value. Therefore, the truth value of $q \in need.p$ is also eventually constant. Finally, as $need.p$ is always a subset of the finite set $nbh0.p$, which is constant while p is at line 26, we can now conclude that the set $need.p$ is eventually constant.

If the set $need.p$ is eventually empty, process p would be eventually always enabled. Weak fairness of p would then imply that process p would leave line 26. Therefore, there is some process q eventually always in $need.p$. We have $q \neq p$ because of $Jq0$ and $Iq0$. This proves

$$(8) \quad Wf(p) \cap \Diamond\Box[p \text{ at } 26] \subseteq \bigcup_{q \neq p} \Diamond\Box[q \in need.p] .$$

Assume that $q \in need.p$ holds from time n_0 onward. We now make a case distinction. First assume $q < p$. If **withdraw**. $q.p > 0$ holds at some time $n \geq n_0$, weak fairness implies that the message **withdraw** will be received at some time $> n_0$, which would falsify $q \in need.p$. Therefore, **withdraw**. $q.p = 0$ holds from time n_0 onward. By predicate $Waq2$, we thus have $q \text{ in } \{26 \dots\}$ and $p \bowtie q$ from time n_0 onward. This proves

$$(9) \quad q < p \Rightarrow Wf(p) \cap \Diamond \Box [q \in need.p] \subseteq \Diamond \Box [q \text{ in } \{26 \dots\} \wedge p \bowtie q] .$$

Next, assume $p < q$. If $\mathbf{gra}.q.p > 0$ holds at some time $n \geq n_0$, this \mathbf{gra} message will be received because of weak fairness, falsifying $q \in need.p$. Therefore, $\mathbf{gra}.q.p = 0$ holds from time n_0 onward. Also by weak fairness, we have $\mathbf{notify}.p.q = \perp$ at some time $n_1 \geq n_0$. Because process p is and remains at line 26, it cannot send such messages again. Therefore, $\mathbf{notify}.p.q = \perp$ holds from time n_1 onward. Because process q sends no \mathbf{gra} message to p after n_0 , weak fairness implies that $dProm(q, p)$ holds infinitely often after n_1 . By *Waq3* this implies that, infinitely often, we have

$$bg(q, p) : \quad q \text{ in } \{27 \dots\} \wedge p \in nbh.q \wedge p \bowtie q .$$

We need the condition $bg(q, p)$ eternally, however, not only infinitely often. For this purpose, we reuse that $vf(q, p)$ is eventually constant, say that $vf(q, p) = k$ holds from time $n_2 \geq n_1$ onward. As condition $bg(q, p)$ contradicts $bf(q, p)$, we have $k > 1$. Therefore, $bf(q, p)$ is false from time n_2 onward. This implies that process q does not execute line 28 from time n_2 onward. Therefore, condition $bg(q, p)$ holds eventually always. Consequently, we obtain

$$(10) \quad p < q \Rightarrow Wf(p) \cap \Diamond \Box [q \in need.p] \subseteq \Diamond \Box [q \text{ in } \{27 \dots\} \wedge p \bowtie q] .$$

5.6 The proof of Theorem 1

Let W be a nonempty set of processes. Let us introduce the abbreviation

$$cf(W) = \bigcup_{q \in W, r \notin W} \Diamond \Box [q \bowtie r] .$$

Now Theorem 1 asserts that $Wf(W) \subseteq \Diamond \Box [p \text{ at } 21] \cup cf(W)$ for every $p \in W$.

We first combine the results of Section 5.5 to prove

$$(11) \quad p \in W \Rightarrow Wf(W) \cap \Diamond \Box [p \text{ at } 26] \subseteq cf(W) .$$

Let xs be an execution in the lefthand set. Because W contains some process that remains eternally at 26, we can consider the lowest process, say $q \in W$, with this property. By formula (8), there is a process $r \neq q$ that remains eternally in $need.q$. If $q < r$, formula (10) implies that r remains eternally in $\{27 \dots\}$ and in conflict with q . By Formula (2), it follows that $r \notin W$ and hence that $xs \in cf(W)$. On the other hand, if $r < q$, Formula (9) implies that r remains eternally in $\{26 \dots\}$ and in conflict with q . If $r \in W$, minimality of q implies that process r proceeds to $\{27, 28\}$, contradicting Formula (2). Therefore $r \notin W$, and $xs \in cf(W)$. This concludes the proof of Formula (11).

By similar arguments, we use Formula (7) to obtain:

$$(12) \quad p \in W \Rightarrow Wf(W) \cap \Diamond \Box [p \text{ at } 25] \subseteq cf(W) .$$

Finally, Theorem 1 follows from the Formulas (1), (2), (5), (6), (11), and (12).

5.7 Progress for lowering

Formula (3) gives progress of the lowering thread at line 33, under the assumption of weak fairness. Progress at line 32, however, requires more than weak fairness. While the lowering thread of p is at line 32, the main thread of p can repeatedly enter and leave the region 22—24. Therefore, the step of line 32 is not eventually always enabled. Indeed, we do not want that resource acquisition suffers for lowering.

We need strong fairness for progress at line 32. Strong fairness at line 32 means that, if process p is infinitely often enabled at line 32, it will eventually take the step of line 32. Formally, for a relation $R \subseteq X^2$, *strong fairness* [16] for R is defined as the set of executions in which R is eventually always disabled or infinitely often taken:

$$sf(R) = Ex \cap (\Diamond \Box [D(R)] \cup \Box \Diamond [R]_2) .$$

If, in some execution, process p is always eventually at line 21, and yet remains at line 32, the step of line 32 is infinitely often enabled, so that indeed strong fairness guarantees that the step will be taken. In combination with formula (3), we thus obtain progress for the lowering thread in the sense that it always returns to its idle state at line 31:

$$\Box \Diamond [p \text{ at } 21] \cap Wf(p) \cap sf(st32(p)) \subseteq \Box \Diamond [p \text{ at } 31] ,$$

where $st32(p)$ is the relation corresponding to the step of p at line 32.

6 Message Complexity and Waiting Times

In the central algorithm, a process exchanges 3 or 4 messages with every neighbour. In the querying phase, at lines 22 and 23, it exchanges 2 messages with every site it is interested in, plus 2 messages for every potential competitor.

In a message passing algorithm, we can distinguish two kinds of waiting. There is waiting for answers that can be sent immediately. Such waiting requires at most 2Δ , because Δ is an upper bound of the time needed for the execution of an alternative plus the time the messages sent are in transit. In the central algorithm, this happens at line 23, for emptiness of *curlist*, and at line 24, for emptiness of *pack* and *wack*. Waiting in the lowering algorithm, at line 33, is also of this kind.

The other kind of waiting is when a process needs to wait for the progress of other processes. These are the important waiting conditions. The central algorithm has two locations where this is the case. At line 25, the process waits for emptiness of *prio* to make accumulation of conflicting processes unlikely. At line 26, it waits for emptiness of *need* to ensure partial mutual exclusion.

The waiting time T_1 for emptiness of *need* at line 26 depends on the conflict graph of the processes that are concurrently in the inner protocol. The middle layer tries to keep this graph small by guaranteeing that conflicting processes do not enter the inner protocol concurrently unless they pass line 25 within a period Δ .

The waiting time T_2 for emptiness of *prio* in line 25 depends on the efficiency of the inner protocol, because for a process p the elements of *prio.p* are in the inner protocol and are removed from *prio.p* when they withdraw. We thus have $T_2 \leq T_1 + \Gamma + \Delta$ where Γ is an upper bound for the time spent in CS. Indeed, every element of *prio* arrives in CS after time T_1 , and at line 28 after $T_1 + \Gamma$, while the message *withdraw* takes time Δ .

The total waiting time for the main loop body is at most $6\Delta + T_2 + T_1 + \Gamma \leq 2T_1 + 2\Gamma + 7\Delta$. This includes 2Δ for concurrent lowering.

The value of T_1 heavily depends on the load of the system and other system parameters: for resource c , the number of jobs activated per Γ that need resource c ; the number of resources per job; the number of resources per site; the number of processes per site; the number of conflicts per job. Of course, all these numbers should be averages, and they are not independent. Experiments are needed to evaluate the performance of the algorithm, and to compare it with other algorithms.

7 Conclusions

The problem of distributed resource allocation is a matter of partial mutual exclusion, with the partiality determined by a dynamic conflict graph. Our solution allows infinitely many processes, and it allows conflicts between every pair of processes. The primary disentanglement is the split into the central algorithm and the registration algorithm.

In the central algorithm, the conflict graph is dynamic but limited by the current registrations, and the jobs can be treated as uninterpreted objects with a compatibility relation. The central algorithm itself consists of three layers. In the outer protocol, the processes communicate their jobs. In the inner protocol they compete for the critical section. The middle layer protects the inner protocol from flooding with conflicting processes.

The neighbourhoods used in the central algorithm are formed in a querying phase in which the processes communicate with a finite number of registration sites. We use a flexible job model that allows e.g. the distinction between read permissions and write permissions. We reach a fully dynamic conflict graph by enabling the processes to modify their registrations.

Our solution does not automatically satisfy the “economy” condition of [4] that permanently tranquil philosophers should not send or receive an infinite number of messages. Indeed, in our algorithm, a permanently idle process that occurs infinitely often in the neighbourhood of other processes will receive and send infinitely many messages. It can avoid this, however, by resetting its registrations to zero.

Our solution is more concurrent than the layered solution of [24]. It satisfies the requirement that, “if a drinker requests a set B of bottles, it should eventually enter its critical region, as long as no other drinker uses or wants any of the bottles in B forever” ([24, p. 243]).

Our algorithm does not minimize the response time. Yet, it may perform reasonably well in this respect, because the middle layer of the central algorithm prohibits entrance for new processes that have known conflicts with processes currently in the inner protocol. In the inner protocol, the lower processes have the advantage that they can force priority over higher conflicting processes. When conflicts in the inner protocol are rare, however, this bias towards the lower processes will not be noticeable.

The algorithm as presented allows several simplifications. (1) The aborting commands of Section 2.8 can be removed. (2) One can decide to give every resource its own site, or to use a single site for all resources. (3) If one takes the simplest job model, i.e. $K = 1$ in Section 2.6, the arrays *fun*, *news*, and *list* reduce to finite sets. (4) One can fix the network topology, i.e., replace the variables *nbh.p* by constants, and remove the registration algorithm. (5) If the set of all processes is a finite and rather small set *Proc*, one can even take *nbh.p* = *Proc* for all *p*.

It would be interesting to see how much the algorithm can be simplified by using reliable synchronous messages, or how the algorithm can be made robust by allowing the asynchronous messages to be lost (and possibly duplicated).

The algorithm could not have been designed without a proof assistant like PVS. This holds in particular for the proofs of safety of registration (the invariants Mq^*), the proof of progress of the central algorithm (Section 5.5), and the use of array *copy* in the registration algorithm to avoid additional waiting.

Acknowledgement. The observation that the algorithm also solves the readers/writers problem was made by Arnold Meijster.

References

- [1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, 1991.
- [2] G.R. Andrews. *Foundations of multithreaded, parallel, and distributed Programming*. Addison Wesley, Reading, etc., 2000.

- [3] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, pages 65–74, St. Louis, 1990.
- [4] K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):623–646, 1984.
- [5] K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison–Wesley, 1988.
- [6] M. Choy and A. Singh. Efficient fault-tolerant algorithms for resource allocation in distributed systems. In *Proc. 24th ACM Symposium on Theory of Computing*, pages 593–602. ACM, 1992.
- [7] P.J. Courtois, F. Heymans, and D.L. Parnas. Concurrent control with “readers” and “writers”. *Commun. ACM*, 14:667–668, 1971.
- [8] A.K. Datta, M. Gradinariu, and M. Raynal. Stabilizing mobile philosophers. *Inf. Process. Lett.*, 95:299–306, 2005.
- [9] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.
- [10] E.W. Dijkstra. A strong P/V-implementation of conditional critical regions. Tech. Rept., Tech. Univ. Eindhoven, EWD 651, see www.cs.utexas.edu/users/EWD, 1977.
- [11] W.H. Hesselink. Partial mutual exclusion for infinitely many processes. Submitted. See <http://arxiv.org/abs/1111.5775>, 2011.
- [12] W.H. Hesselink. Distributed resource allocation. <http://www.cs.rug.nl/~wim/mechver/distrRscAlloc>, 2012.
- [13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [14] G.J. Holzmann. *The SPIN Model Checker, primer and reference manual*. Addison-Wesley, 2004.
- [15] L. Lamport. A new solution of Dijkstra’s concurrent programming problem. *Commun. ACM*, 17:453–455, 1974.
- [16] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, 1994.
- [17] E.A. Lycklama and V. Hadzilacos. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Trans. Program. Lang. Syst.*, 13:558–576, 1991.
- [18] N.A. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal of Computer and System Sciences*, 23:254–278, 1981.
- [19] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufman, San Francisco, 1996.
- [20] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference*, 2001. <http://pvs.csl.sri.com>
- [21] I. Page, T. Jacob, and E. Chern. Fast algorithms for distributed resource allocation. *IEEE Trans. Parallel and Distributed Systems*, 4:188–197, 1993.

- [22] I. Rhee. A modular algorithm for resource allocation. *Distr. Comput.*, 11:157–168, 1998.
- [23] P.A.G. Sivilotti, S.M. Pike, and N. Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. Ohio State University, 2000.
- [24] J.L. Welch and N.A. Lynch. A modular drinking philosophers algorithm. *Distr. Comput.*, 6:233–244, 1993.